# MAPSI

Modèles et Algorithmes de Probabilités et Statistiques pour l'Informatique



# Rappels de Probas / stats

## La planche de Galton

### 1. Loi de Bernoulli

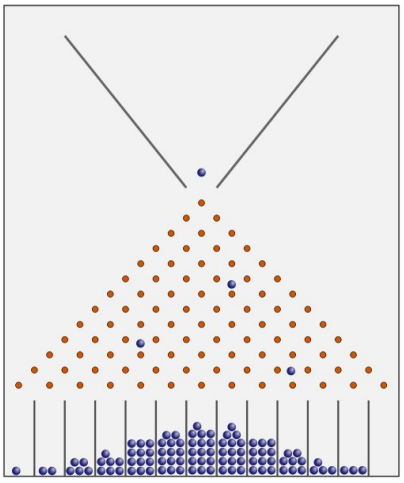
Ecrivez une fonction **bernoulli** qui prend en argument un paramètre **p** et renvoie 1 (succès) avec la probabilité **p** et 0 (échec) avec la probabilité **1-p**.

### 2. Loi binomiale

Ecrivez une fonction **binomiale** qui prend en argument un paramètre **n** et un paramètre **p** et qui renvoie un nombre tiré aléatoirement selon la distribution binomiale $\mathcal{B}(n,p)$.

### 3. Histogramme de la loi binomiale

Dans cette question, on considère une planche de Galton de hauteur **n**. On rappelle que des bâtons horizontaux (oranges) sont cloués à cette planche comme le montre la figure ci-dessous. Des billes bleues tombent du haut de la planche et, à chaque niveau, se retrouvent à la verticale d'un des bâtons. Elles vont alors tomber soit à gauche, soit à droite du bâton, jusqu'à atteindre le bas de la planche. Ce dernier est constitué de petites boites dont les bords sont symbolisés par les lignes verticales grises. Chaque boite renferme des billes qui sont passées exactement le même nombre de fois à droite des bâtons oranges. Par exemple, la boite la plus à gauche renferme les billes qui ne sont jamais passées à droite d'un bâton, celle juste à sa droite renferme les billes passées une seule fois à droite d'un bâton et toutes les autres fois à gauche, et ainsi de suite.

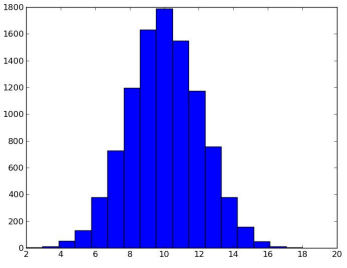


La répartition des billes dans les boites suit donc une loi binomiale $\mathcal{B}(n,0.5)$. Ecrivez un script qui crée un **array** de 1000 cases dont le contenu correspond à 1000 instanciations de la loi binomiale $\mathcal{B}(n,0.5)$. Afin de voir la répartition des billes dans la planche de Galton, tracez l'histogramme de ce tableau. Vous pourrez utiliser la fonction **hist** de **matplotlib.pyplot**:

```
import matplotlib.pyplot as plt
plt.hist ( tableau_1000_cases, nb_bins)
```

[$[Get Code]]

Pour le nombre de bins, calculez le nombre de valeurs différentes dans votre tableau.




## Visualisation d'indépendances

### 1. Loi normale centrée

On souhaite visualiser la fonction de densité de la loi normale. Pour cela, on va créer un ensemble de **k** points $(x_i,y_i)$, pour des $x_i$ équi-espacés variant de $-2\sigma$ à $2\sigma$, les $y_i$ correspondant à la valeur de la fonction de densité de la loi normale centrée de variance $\sigma^2$, autrement dit $\mathcal{N}(0,\sigma^2)$. Ecrivez une fonction **normale** qui, étant donné un paramètre **k** impair et un paramètre **sigma** renvoie l'array des **k** valeurs $y_i$. Afin que l'array soit bien symmétrique, on

SEARCH
Go

TUTORIEL NUMPY

INFOS COURS/TD/TME

SEMAINIER

Semaine 1
Semaine 2
Semaine 3
Semaine 4
Semaine 5
Rapport 1
Semaine 6
Semaine 7?
Semaine 8
Semaine 9?
Semaine 10?

LIENS

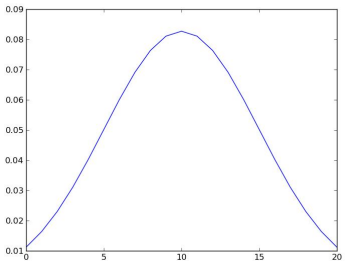
edit SideBar

lèvera une exception si **k** est pair:

```
def normale ( k, sigma ):
    if k % 2 == 0:
        raise ValueError ( 'le nombre k doit etre impair' )
    .....
```

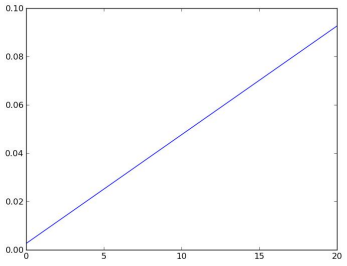
[$[Get Code]]

Vous vérifierez la validité de votre fonction en affichant grâce à la fonction **plot** les points générés dans une figure.



## 2. Distribution de probabilité affine

Dans cette question, on considère une généralisation de la distribution uniforme: une distribution affine, c'est-à-dire que la fonction de densité est une droite, mais pas forcément horizontale, comme le montre la figure ci-dessous:



Ecrivez une fonction **proba_affine** qui, comme dans la question précédente, va générer un ensemble de $k$ points $y_i$ , $i = 0, \ldots, k-1$, représentant cette distribution (paramétrée par sa pente **slope**). On supposera que $k$ est impair. Si la pente est égale à 0, c'est-à-dire si la distribution est uniforme, chaque point $y_i$ devrait être égal à $1/k$ (afin que la somme des $y_i$ soit égale à 1). Si la pente est différente de 0, il suffit de choisir, pour tout $i = 0, \ldots, k-1$,

$$y_i = \frac{1}{k} + \left( i - \frac{k-1}{2} \right) \times slope$$

Vous pourrez aisément vérifier que la somme des $y_i$ est alors éagle à 1. Afin que la distribution soit toujours positive (c'est quand même un minimum pour une distribution de probabilité), il faut que la pente **slope** ne soit ni trop grande ni trop petite. Le bout de code ci-dessous lèvera une exception si la pente est trop élevée et indiquera la pente maximale possible.

```
def proba_affine ( k, slope ):
    if k % 2 == 0:
        raise ValueError ( 'le nombre k doit etre impair' )
    if abs ( slope  ) > 2 / ( k * k ):
        raise ValueError ( 'la pente est trop raide : pente max = ' +
        str ( 2 / ( k * k ) ) )
    .....
```

[$[Get Code]]

## 3. Distribution jointe

Ecrivez une fonction **Pxy** qui, étant donné deux tableaux à 1 dimension générés par les fonctions des questions précédentes et représentant deux distributions de probabilités $P(A)$ et $P(B)$, renvoie la distribution jointe $P(A, B)$ sous forme d'un tableau à 2 dimensions, en supposant que $A$ et $B$ sont des variables aléatoires indépendantes. Par exemple, si:

```
PA = np.array ( [0.2, 0.7, 0.1] )
PB = np.array ( [0.4, 0.4, 0.2] )
```

[$[Get Code]]

alors **Pxy** ( PA,PB ) renverra le tableau :

```
array([[ 0.08,  0.08,  0.04],
       [ 0.28,  0.28,  0.14],
       [ 0.04,  0.04,  0.02]])
```

[$[Get Code]]

## 4. Affichage de la distribution jointe

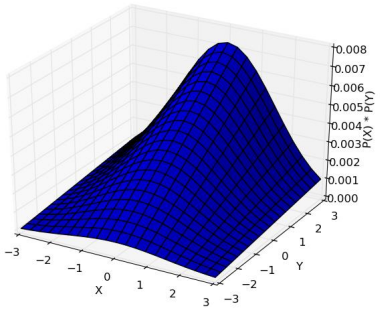
Le code ci-dessous permet d'afficher en 3D une probabilité jointe générée par la fonction précédente. Exécutez-le avec une probabilité jointe résultant de la combinaison d'une loi normale et d'une distribution affine. Si le contenu de la fenêtre est vide, redimensionnez celle-ci et le contenu devrait apparaître. Cliquez à la souris à l'intérieur de la fenêtre et bougez la souris en gardant le bouton appuyé afin de faire pivoter la courbe. Observez sous différents angles cette courbe. Refaites l'expérience avec une probaiblité jointe résultant de deux lois normales. Essayez de comprendre ce que signifie, visuellement, l'indépendance probabiliste. Vous pouvez également recommencer l'expérience avec le logarithme des lois jointes.

```
from mpl_toolkits.mplot3d import Axes3D

def dessine ( P_jointe ):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
```

```
    x = linspace ( -3, 3, P_jointe.shape[0] )
    y = linspace ( -3, 3, P_jointe.shape[1] )
    X, Y = np.meshgrid(x, y)
    ax.plot_surface(X, Y, P_jointe, rstride=1, cstride=1 )
    ax.set_xlabel('A')
    ax.set_ylabel('B')
    ax.set_zlabel('P(A) * P(B)')
    plt.show ()
```

[$[Get Code]]




# Indépendances conditionnelles et consommation mémoire

Le but de cet exercice est d'exploiter les probabilités conditionnelles et les indépendances conditionnelles afin de décomposer une probabilité jointe en un ensemble de "petites probabilités conditionnelles". Cela permet de stocker des probabilités jointes de grandes tailles.

Pour simplifier, nous allons considérer un ensemble $X_0, \ldots, X_n$ de variables aléatoires binaires (elles ne peuvent prendre que 2 valeurs : 0 et 1). Ainsi, une manière de représenter la distribution $P(X_0, \ldots, X_n)$ est d'utiliser un **array** numpy $\mathbb{P}$ de taille $2^{n+1}$ de telle sorte que :

$$P(X_0 = x_1, \ldots, X_n = x_n) = \mathbb{P}[\sum_{i=0}^{n} x_i \times 2^i].$$

Autrement dit, pour tout index $j \in \{0, \ldots, 2^n\}$ du tableau $\mathbb{P}$, si l'on traduit $j$ en binaire, le bit le plus à droite correspond à la valeur de la variable $X_0$, le 2ème bit le plus à droite correspond à $X_1$, et ainsi de suite. Par exemple, $\mathbb{P}$ = [0.05, 0.1, 0.15, 0.2, 0.02, 0.18, 0.13, 0.17] correspond à la distribution :

| $X_2 = 0$ | | | | $X_2 = 1$ | | | |
|---|---|---|---|---|---|---|---|
| $X_1 = 0$ | | $X_1 = 1$ | | $X_1 = 0$ | | $X_1 = 1$ | |
| $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ |
| 0.05 | 0.1 | 0.15 | 0.2 | 0.02 | 0.18 | 0.13 | 0.17 |

## 1. Préparation de l'exercice : marginalisation

La définition d'une probabilité conditionnelle est, pour tout triplet de variables aléatoires ou d'ensembles disjoints de variables aléatoires $A, B, C$, $P(A|B) = \frac{P(A,B)}{P(B)}$. Or, $P(B) = \sum_A P(A, B)$. Ici, nous allons calculer $P(B)$ à partir de $P(A, B)$. Le code ci-dessous permet de réaliser la sommation sur une seule variable représentée par son index $i$ dans l'ensemble de variables $X_0, \ldots, X_i, \ldots, X_n$. Ainsi, l'application de **project1Var** sur $\mathbb{P}$ = [0.05, 0.1, 0.15, 0.2, 0.02, 0.18, 0.13, 0.17] avec l'indice 1 ($X_1$) produira:

| $X_2 = 0$ | | $X_2 = 1$ | |
|---|---|---|---|
| $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ |
| 0.2 | 0.3 | 0.15 | 0.35 |

```
def project1Var ( P, index ):
    """
    supprime 1 variable d'une probabilité jointe

    Param P : une distribution de proba jointe sous forme d'un array à 1
        dimension ( toutes les variables aléatoires sont supposées binaires )
    Param index : représente la variable aléatoire à marginaliser
        (0 = 1ère variable, 1 = 2ème variable, etc).
    """
    length = 2**( index + 1 )
    reste = 2**index
    vect = np.zeros ( P.size / 2 )
    for i in range ( P.size ):
        j = floor ( i / length ) * length / 2 + ( i % reste )
        vect[j] += P[i]
    return vect
```

[$[Get Code]]

Dans la suite, il pourra être utile de projeter plusieurs variables. La fonction ci-dessous permet de réaliser cette opération en passant un array numpy d'indices:

```
def project ( P, ind_to_remove ):
    """
    Calcul de la projection d'une distribution de probas

    Param P une distribution de proba sous forme d'un array à 1 dimension
    Param ind_to_remove un array d'index representant les variables à
    supprimer. 0 = 1ère variable, 1 = 2ème var, etc.
    """
    v = P
    ind_to_remove.sort ()
    for i in range ( ind_to_remove.size - 1, -1, -1 ):
        v = project1Var ( v, ind_to_remove[i] )
    return v
```

[$[Get Code]]

## 2. Préparation de l'exercice : restauration des indices

L'inconvénient des fonctions **project** et **project1Var** est qu'elles modifient la taille et l'indexation des tableaux. Or

cela n'est pas souhaitable si l'on veut utiliser les opérateurs * et / des tableaux numpy (si l'on veut additionner 2 tableaux t1 et t2 de même taille, il suffit de faire t1 + t2). Nous allons maintenant faire en sorte que les tailles et les index des tableaux que nous allons manipuler restent inchangés par projection. Pour cela, la fonction **expanse1Var** ci-dessous, réexpanse comme il se doit une probabilité projetée par **project1Var**. Par exemple, si on applique **project1Var** avec l'indice 1 sur :

| $X_2 = 0$ | | | | $X_2 = 1$ | | | |
|---|---|---|---|---|---|---|---|
| $X_1 = 0$ | | $X_1 = 1$ | | $X_1 = 0$ | | $X_1 = 1$ | |
| $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ |
| 0.05 | 0.1 | 0.15 | 0.2 | 0.02 | 0.18 | 0.13 | 0.17 |

on obtient:

| $X_2 = 0$ | | $X_2 = 1$ | |
|---|---|---|---|
| $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ |
| 0.2 | 0.3 | 0.15 | 0.35 |

et si on applique **expanse1Var** sur ce tableau, toujours avec l'indice 1, on obtient :

| $X_2 = 0$ | | | | $X_2 = 1$ | | | |
|---|---|---|---|---|---|---|---|
| $X_1 = 0$ | | $X_1 = 1$ | | $X_1 = 0$ | | $X_1 = 1$ | |
| $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ | $X_0 = 0$ | $X_0 = 1$ |
| 0.2 | 0.3 | 0.2 | 0.3 | 0.15 | 0.35 | 0.15 | 0.35 |

qui représente une copie de $P(X_0, X_2)$ pour chaque valeur de $X_1$.

```
def expanse1Var ( P, index ):
    """
    duplique une distribution de proba |X| fois, où X est une des variables
    aléatoires de la probabilité jointe P. Les variables étant supposées
    binaires, |X| = 2. La duplication se fait à l'index de la variable passé
    en argument.
    Par exemple, si P = [0,1,2,3] et index = 0, expanse1Var renverra
    [0,0,1,1,2,2,3,3]. Si index = 1, expanse1Var renverra [0,1,0,1,2,3,2,3].

    Param P : une distribution de proba sous forme d'un array à 1 dimension
    Param index : représente la variable à dupliquer (0 = 1ère variable,
       1 = 2ème variable, etc).
    """
    length = 2**(index+1)
    reste = 2**index
    vect = np.zeros ( P.size * 2 )
    for i in range ( vect.size ):
        j = floor ( i / length ) * length / 2 + ( i % reste )
        vect[i] = P[j]
    return vect
```

[$[Get Code]]

A l'instar de la fonction **project**, nous proposons ici une fonction **expanse** qui permet d'expanser plusieurs variables :

```
def expanse ( P, ind_to_add ):
    """
    Expansion d'une probabilité projetée

    Param P une distribution de proba sous forme d'un array à 1 dimension
    Param ind_to_add un array d'index representant les variables permettant
    de dupliquer la proba P. 0 = 1ère variable, 1 = 2ème var, etc.
    """
    v = P
    ind_to_add.sort ()
    for ind in ind_to_add.size:
        v = expanse1Var ( v, ind )
    return v
```

[$[Get Code]]

## 3. Probabilités conditionnelles

Ecrivez une fonction **proba_conditionnelle** ( P ) qui, étant donné une probabilité jointe P sur un ensemble de variables $X_0, \ldots, X_n$, renvoie le tableau numpy correspondant à la probabilité conditionnelle $P(X_n|X_0, \ldots, X_{n-1})$. Vous pourrez vous aider de la fonction ci-dessous qui calcule le nombre de variables (ici $n+1$) sur lesquelles porte la probabilité jointe P:

```
def nb_vars ( P ):
    i = P.size
    nb = 0
    while i > 1:
        i /= 2
        nb += 1
    return nb
```

[$[Get Code]]

## 4. Indépendances conditionnelles

Ecrivez une fonction **is_indep** ( P, index, epsilon ) qui, étant donné une distribution de probabilité jointe $P$ sur un ensemble de variables $X_0, \ldots, X_n$ et l'indice $i$ d'une variable $X_i$ différente de $X_n$, renvoie un booléen indiquant si $X_n$ est indépendant de $X_i$ conditionnellement aux autres variables aléatoires. Vous pourrez avantageusement exploiter le fait qu'une telle indépendance existe si et seulement si :

$$P(X_n|X_0, \ldots, X_{n-1}) = P(X_n|X_0, \ldots, X_{i-1}, X_{i+1}, \ldots, X_{n-1}).$$

Etant donné que l'arithmétique flotante sur ordinateur n'a pas une précision infinie, il peut y avoir des erreurs d'arrondis dans vos calculs. Aussi, plutôt que de tester l'égalité entre les deux formules, il est préférable de tester qu'en valeur absolue, leur différence reste inférieure à un petit epsilon (ici, vous utiliserez epsilon = 1e-6).

Afin de tester votre fonction, vous pourrez charger à l'aide de np.loadtxt le fichier suivant : **asia**, qui contient une probabilité jointe sur un ensemble de 8 variables aléatoires. Ce fichier provient du site web suivant :

**http://www.cs.huji.ac.il/site/labs/compbio/Repository/networks.html**

Il représente un problème de diagnostic d'une maladie respiratoire (dyspnée) en fonction d'autres maladies (turberculose, cancer du poumon) ainsi que de facteurs déclenchants de ces maladies et d'examens médicaux permettant de les discriminer :

| $X_0$ | visit_to_Asia? |
|---|---|

| | |
|---|---|
| $X_1$ | tuberculosis? |
| $X_2$ | tuberculos_or_cancer? |
| $X_3$ | positive_Xray? |
| $X_4$ | lung_cancer? |
| $X_5$ | smoking? |
| $X_6$ | bronchitis? |
| $X_7$ | dyspnoea? |

## 5. Exploitation d'indépendances conditionnelles

En exploitant itérativement la fonction précédente sur toutes les variables $X_{n-1}, \ldots, X_0$, on peut réduire significativement la taille de la probabilité conditionnelle et donc optimiser son stockage. Dans cette question, vous allez écrire une fonction **find_indep** ( P, epsilon ) qui, étant donné une probabilité jointe $P(X_0, \ldots, X_n)$ calcule la probabilité conditionnelle $P(X_n|X_0, \ldots, X_{n-1})$ puis supprime itérativement toutes les variables $X_i$ indépendantes de $X_n$ conditionnellement aux autres variables aléatoires. Par exemple, pour une probabilité jointe $P(X_0, X_1, X_2, X_3)$, on commence par calculer $P(X_3|X_0, X_1, X_2)$. Si $X_2$ est indépendant de $X_3$ conditionnellement à $(X_0, X_1)$, alors $P(X_3|X_0, X_1, X_2) = P(X_3|X_0, X_1)$. On peut alors tester si $X_1$ est indépendant de $X_3$ conditionnellement à $X_0$ (autrement dit, on effectue nos tests d'indépendance à partir de la probabilité conditionnelle la plus petite, la plus compacte). S'il n'y a pas indépendance, on teste l'indépendance de $X_0$ et de $X_3$ conditionnellement à $X_1$ (toutes les variables de conditionnement excepté $X_0$). S'il y a bien indépendance alors $P(X_3|X_0, X_1) = P(X_3|X_1)$ et donc $P(X_3|X_0, X_1, X_2) = P(X_3|X_1)$. Votre fonction **find_indep** renverra un triplet constitué du nombre de variables dans la probabilité jointe passée en argument, de la probabilité conditionnelle la plus compacte que vous avez calculée, et de la liste des variables qui constituent cette probabilité conditionnelle). **Attention:** pour que la suite soit correcte, il faut tester les indépendances conditionnelles dans l'ordre $X_{n-1}$ jusqu'à $X_0$.

Testez votre fonction sur la probabilité jointe de "asia".

## 6. Expression compacte d'une probabilité jointe

Vous avez vu en cours que $P(A, B) = P(A|B)P(B)$, et ce quel que soient les ensembles de variables aléatoires disjoints $A$ et $B$. En posant $A = X_n$ et $B = \{X_0, \ldots, X_{n-1}\}$, on obtient donc:

$$P(X_0, \ldots, X_n) = P(X_n|X_0, \ldots, X_{n-1})P(X_0, \ldots, X_{n-1}).$$

On peut réitérer cette opération pour le terme de droite en posant $A = X_{n-1}$ et $B = \{X_0, \ldots, X_{n-1}\}$, et ainsi de suite. Donc, par récurrence, on a:

$$P(X_0, \ldots, X_n) = P(X_0) \times \prod_{i=1}^{n} P(X_i|X_0, \ldots, X_{i-1}).$$

On peut alors appliquer la fonction **find_indep** pour chaque probabilité de ce produit afin de réduire sa taille. Vous allez donc écrire une fonction **find_all_indep** ( P, epsilon ) qui, étant donné une probabilité jointe $P(X_0, \ldots, X_n)$ détermine chaque probabilité jointe $P(X_0, \ldots, X_i) = \sum_{X_{i+1}, \ldots, X_n} P(X_0, \ldots, X_n)$ et applique **find_indep** sur cette probabilité afin de produire une probabilité conditionnelle de petite taille. Vous afficherez, pour chaque valeur de $i$, le nombre de variables de la distribution $P(X_0, \ldots, X_i)$ ainsi que celui de la probabilité conditionnelle compacte que vous aurez calculée. Vous afficherez également la consommation mémoire totale de vos probabilités conditionnelles compactes et la comparerez avec celle de $P(X_0, \ldots, X_n)$.

Enfin, vous calculerez le produit de $P(X_0)$ et de vos probabilités conditionnelles compactes. Si vous ne vous êtes pas trompé(e), vous devriez obtenir une copie de la probabilité jointe initiale $P(X_0, \ldots, X_n)$ (à cause des erreurs d'arrondis, vous testerez juste que la valeur absolue de la différence entre $P(X_0, \ldots, X_n)$ et votre produit est inférieur à epsilon).

## 7. Applications pratiques

La technique de décomposition que vous avez vue est effectivement utilisée en pratique. Vous pouvez voir le gain obtenu sur différentes distributions de probabilité du site :

**http://www.cs.huji.ac.il/site/labs/compbio/Repository/networks.html**

Cliquez sur le nom du dataset que vous voulez visualiser et téléchargez son .bif ou .dsl. Afin de visualiser le contenu du fichier, vous allez utiliser la librairie aGrUM et son wrapper python : pyAgrum. Ce dernier est installé dans /Vrac sur les machines de la ppti et il faut donc indiquer à python où il se trouve. Pour cela, il faut ajouter les 2 lignes suivantes dans le fichier .bashrc de votre home directory:

```
export LD_LIBRARY_PATH=/Vrac/agrum/lib/python2.7/site-packages/pyAgrum:$LD_LIBRARY_PATH
export PYTHONPATH=/Vrac/agrum/lib/python2.7/site-packages/:$PYTHONPATH
```

**[$[Get Code]]**

Le fichier .bashrc étant lu à l'ouverture de vos consoles, il faut maintenant que vous ouvriez une nouvelle console dans laquelle vous exécuterez spyder.

Si vous souhaitez télécharger la librairie pour l'installer chez vous, vous pouvez télécharger une des deux archives suivantes : **aGrUM.tgz** ou **aGrUM.zip**. Pour désarchiver le .tgz, il vous faudra taper la commande `tar xvzf aGrUM.tgz` et pour dézipper le .zip, il faudra taper `unzip aGrUM.zip`.

Ensuite, pour installer la librairie, allez dans le répertoire aGrUM et tapez `./act install -d installation_path pyAgrum` pour installer aGrUM dans le répertoire `installation_path` de votre choix. La page **forge aGrUM** contient plus d'informations sur la librairie. Les deux lignes d'environnement LD_LIBRARY_PATH et PYTHONPATH ci-dessus doivent alors être mises à jour en fonction du répertoire que vous avez choisi. Si vous avez choisi /home/toto/usr, alors les lignes à saisir dans votre .bashrc sont (très probablement) :

```
export LD_LIBRARY_PATH=/home/toto/usr/lib/python2.7/site-packages/pyAgrum:$LD_LIBRARY_PATH
export PYTHONPATH=/home/toto/usr/lib/python2.7/site-packages/:$PYTHONPATH
```

**[$[Get Code]]**

Le code suivant vous permettra alors de visualiser votre dataset: la valeur indiquée après "domainSize" est la taille de la probabilité jointe d'origine (en nombre de flotants) et celle après "parameters" est la taille de la probabilité sous forme compacte (somme des tailles des probabilités conditionnelles compactes).

```
# chargement de la librairie aGrUM
import pyAgrum as gum
```

```
# chargement du fichier bif ou dsl
bn = gum.loadBN ( "nom du fichier" )

# affichage de la taille des probabilités jointes compacte et non compacte
print bn
```

[$[Get Code]]